Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Computer Architecture – ENCS4370

Project II
RISC Processor Design and Verification

Prepared by:

Ahmad Hamdan - 1210241

Mohammad Fareed - 1212387

Qossay Rida - 1211553

Instructor: Aziz Qaroush

Section: 1

Date: 19/6/2024

Abstract

This project involves designing and verifying a simple pipelined RISC processor in Verilog, featuring a five-stage pipeline: fetch, decode, ALU, memory access, and write-back. We implemented an instruction set architecture (ISA) with R-type, I-type, J-type, and S-type instructions, supported by 8 general-purpose registers and byte-addressable memory. Our report provides a comprehensive description of the datapath, control path, control signals, and verification methods, including detailed simulation results. We demonstrate the processor's functionality through rigorous testing and analysis.

Table of Contents

Table of Figures

Design and Implementation

Introduction

For an optimal solution that won’t take as much time and without consecutive problems, we had to approach the project carefully. First, **we decided to implement a pipelined processor**. Then, we started with each instruction type creating the datapath and control path needed by adding functional units or simply editing the datapath. **We edited the I-Type instruction format after discussing it with Dr. Aziz by moving the mode bit after the registers.**

R-Type	Opcode^4	Rd^3	Rs1^3	Rs2^3	Unused^3
I-Type	Opcode^4	Rd^3	Rs1^3	M^1	Immediate^5
J-Type	Opcode^4	Jump Offset^{12} / Unused^{12}			
S-Type	Opcode^4	Rs^3		Immediate^9	

After designing the datapath, we started with writing the verilog description code for the processor.

Each component in the datapath was built separately in the code and then connected with it’s stage. Then all stages are called in the final module so all stages work in parallel.

Since we went for the pipelined processor implementation, we needed to build a hazard detector and forward which will handle any hazards occuring.

RTL

R-Type

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch operands:** data1 ← Reg(Rs1), data2 ← Reg(Rs2)
- **Execute operation:** ALU_result ← data1 (+/-/&) data2
- **Write ALU result:** Reg(Rd) ← ALU_result
- **Next PC address:** PC ← PC + 2

ANDI & ADDI

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch operands:** data1 ← Reg(Rs1), immediate ← Extend(imm)
- **Execute operation:** ALU_result ← data1 (&/+) immediate
- **Write ALU result:** Reg(Rd) ← ALU_result
- **Next PC address:** PC ← PC + 2

LW

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch base register:** base ← Reg(Rs1)
- **Calculate address:** address ← base + Extend(imm)
- **Read memory:** data ← MEM[address]
- **Write register Rt:** Reg(Rd) ← data
- **Next PC address:** PC ← PC + 2

LBu

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch base register:** base ← Reg(Rs1)
- **Calculate address:** address ← base + Extend(imm)
- **Read memory:** data ← MEM[address]
- **Write register Rt:** Reg(Rd) ← zero_extend(data)
- **Next PC address:** PC ← PC + 2

LBs

- **Fetch instruction:** Instruction ← MEM[PC]

- **Fetch base register:** base ← Reg(Rs1)
- **Calculate address:** address ← base + Extend(imm)
- **Read memory:** data ← MEM[address]
- **Write register Rt:** Reg(Rd) ← sign_extend(data)
- **Next PC address:** PC ← PC + 2

SW

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch registers:** base ← Reg(Rs1), data ← Reg(Rd)
- **Calculate address:** address ← base + Extend(imm)
- **Write memory:** MEM[address] ← data
- **Next PC address:** PC ← PC + 2

BGT

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch operands:** data1 ← Reg(Rd), data2 ← Reg(Rs1)
- **Branch decision:** if (data1 > data2) then PC ← PC + sign_ext(imm) else PC ← PC + 2

BLT

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch operands:** data1 ← Reg(Rd), data2 ← Reg(Rs1)
- **Branch decision:** if (data1 < data2) then PC ← PC + sign_ext(imm) else PC ← PC + 2

BEQ

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch operands:** data1 ← Reg(Rd), data2 ← Reg(Rs1)
- **Branch decision:** if (data1 == data2) then PC ← PC + sign_ext(imm) else PC ← PC + 2

BNE

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch operands:** data1 ← Reg(Rd), data2 ← Reg(Rs1)
- **Branch decision:** if (data1 != data2) then PC ← PC + sign_ext(imm) else PC ← PC + 2

JMP

- **Fetch instruction:** Instruction ← MEM[PC]
- **Calculate target address:** target ← {PC[15:10], Immediate}

- **Jump:** PC ← target

CALL

- **Fetch instruction:** Instruction ← MEM[PC]
- **Calculate target address:** target ← {PC[15:10], Immediate}
- **Save return address:** Reg(R15) ← PC + 2
- **Jump:** PC ← target

RET

- **Fetch instruction:** Instruction ← MEM[PC]
- **Return:** PC ← Reg(R7)

SV

- **Fetch instruction:** Instruction ← MEM[PC]
- **Store value:** MEM[Reg(Rs)] ← imm
- **Next PC address:** PC ← PC + 2

The Datapath

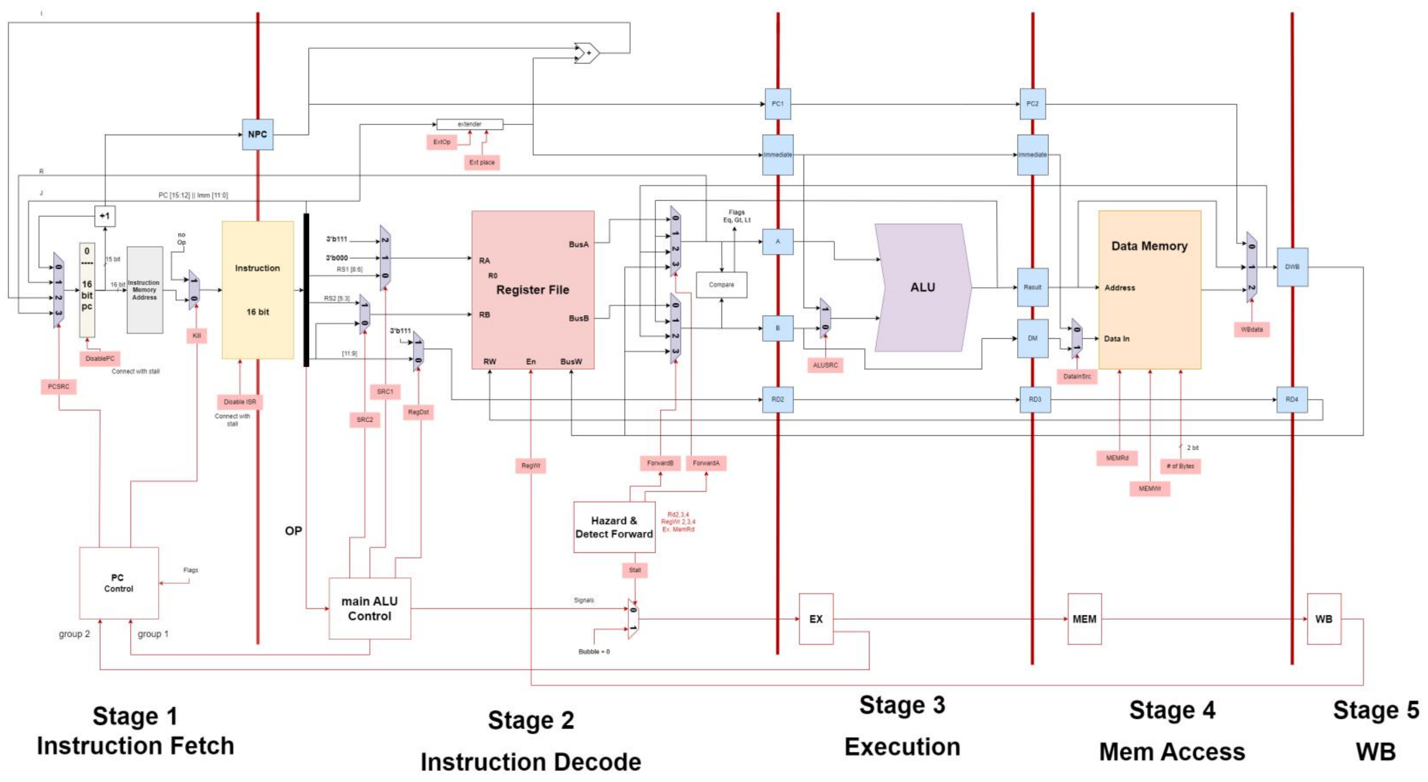
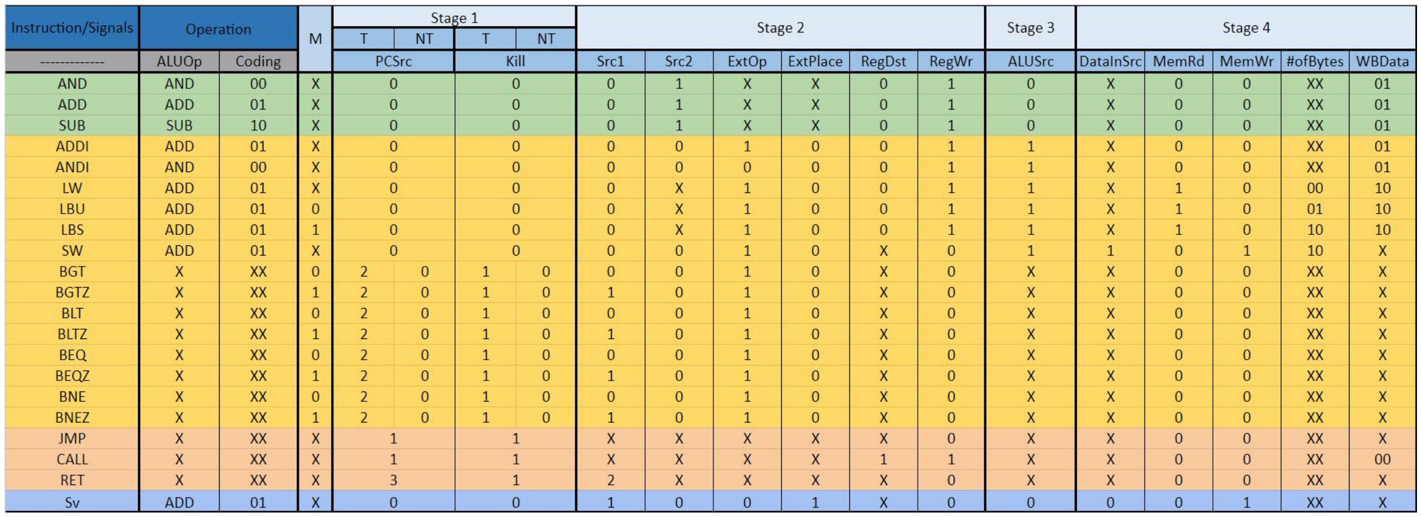


Figure 1 The datapath

Some key points concerning the datapath:

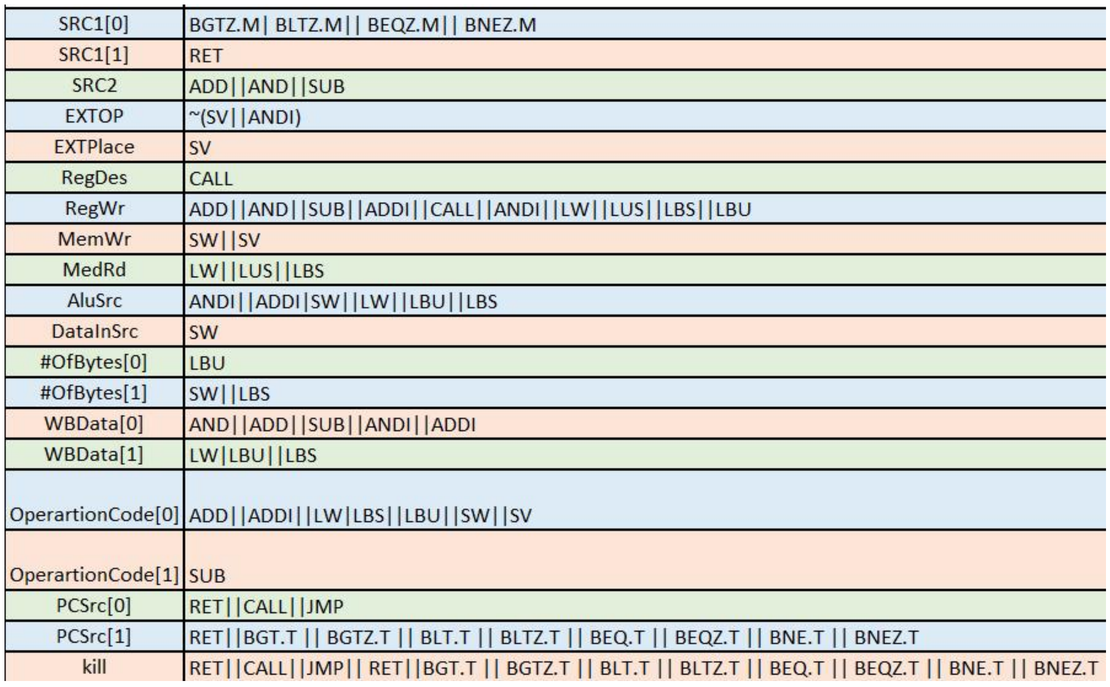
- The control units and control signals lines are colored red to be distinct.
- The 16-bit PC has 4 possible inputs:
    - The output A of the decode stage which is R7 when the instruction is Return.
    - PC + 2 + extended immediate when the instruction is I-Type and the branch is taken.
    - PC[15:10] concatenated with the immediate when the instruction is Jump.
    - PC + 2 otherwise.
- In case of a data dependency, we need to have a stall cycle to handle it, we do the following:
    - Disable the PC so it's value does not change therefor we will fetch the same instruction again
- In case of an instruction that might change the PC such as jump or branch, we do the following:
    - We kill the fetched instruction after the PC changing instruction, replacing it with a dummy instruction (R1 = R1 + R0), because we determine what's the next PC at the end of the decode stage.

Control Signals



Instruction/Signals	Operation		M	Stage 1				Stage 2						Stage 3	Stage 4				
				T	NT	T	NT												
-------------	ALUOp	Coding		PCSrc		Kill		Src1	Src2	ExtOp	ExtPlace	RegDst	RegWr	ALUSrc	DataInSrc	MemRd	MemWr	#ofBytes	WBData
AND	AND	00	X	0		0		0	1	X	X	0	1	0	X	0	0	XX	01
ADD	ADD	01	X	0		0		0	1	X	X	0	1	0	X	0	0	XX	01
SUB	SUB	10	X	0		0		0	1	X	X	0	1	0	X	0	0	XX	01
ADDI	ADD	01	X	0		0		0	0	1	0	0	1	1	X	0	0	XX	01
ANDI	AND	00	X	0		0		0	0	0	0	0	1	1	X	0	0	XX	01
LW	ADD	01	X	0		0		0	X	1	0	0	1	1	X	1	0	00	10
LBU	ADD	01	0	0		0		0	X	1	0	0	1	1	X	1	0	01	10
LBS	ADD	01	1	0		0		0	X	1	0	0	1	1	X	1	0	10	10
SW	ADD	01	X	0		0		0	0	1	0	X	0	1	1	0	1	10	X
BGT	X	XX	0	2	0	1	0	0	0	1	0	X	0	X	X	0	0	XX	X
BGTZ	X	XX	1	2	0	1	0	1	0	1	0	X	0	X	X	0	0	XX	X
BLT	X	XX	0	2	0	1	0	0	0	1	0	X	0	X	X	0	0	XX	X
BLTZ	X	XX	1	2	0	1	0	1	0	1	0	X	0	X	X	0	0	XX	X
BEQ	X	XX	0	2	0	1	0	0	0	1	0	X	0	X	X	0	0	XX	X
BEQZ	X	XX	1	2	0	1	0	1	0	1	0	X	0	X	X	0	0	XX	X
BNE	X	XX	0	2	0	1	0	0	0	1	0	X	0	X	X	0	0	XX	X
BNEZ	X	XX	1	2	0	1	0	1	0	1	0	X	0	X	X	0	0	XX	X
JMP	X	XX	X	1		1		X	X	X	X	X	0	X	X	0	0	XX	X
CALL	X	XX	X	1		1		X	X	X	X	1	1	X	X	0	0	XX	00
RET	X	XX	X	3		1		2	X	X	X	X	0	X	X	0	0	XX	X
Sv	ADD	01	X	0		0		1	0	0	1	X	0	0	0	0	1	XX	X

Figure 2 Control Signals



SRC1[0]	BGTZ.M| BLTZ.M|| BEQZ.M|| BNEZ.M
SRC1[1]	RET
SRC2	ADD||AND||SUB
EXTOP	~(SV||ANDI)
EXTPlace	SV
RegDes	CALL
RegWr	ADD||AND||SUB||ADDI||CALL||ANDI||LW||LUS||LBS||LBU
MemWr	SW||SV
MedRd	LW||LUS||LBS
AluSrc	ANDI||ADDI|SW||LW||LBU||LBS
DataInSrc	SW
#OfBytes[0]	LBU
#OfBytes[1]	SW||LBS
WBData[0]	AND||ADD||SUB||ANDI||ADDI
WBData[1]	LW|LBU||LBS
OperartionCode[0]	ADD||ADDI||LW|LBS||LBU||SW||SV
OperartionCode[1]	SUB
PCSrc[0]	RET||CALL||JMP
PCSrc[1]	RET||BGT.T || BGTZ.T || BLT.T || BLTZ.T || BEQ.T || BEQZ.T || BNE.T || BNEZ.T
kill	RET||CALL||JMP|| RET||BGT.T || BGTZ.T || BLT.T || BLTZ.T || BEQ.T || BEQZ.T || BNE.T || BNEZ.T

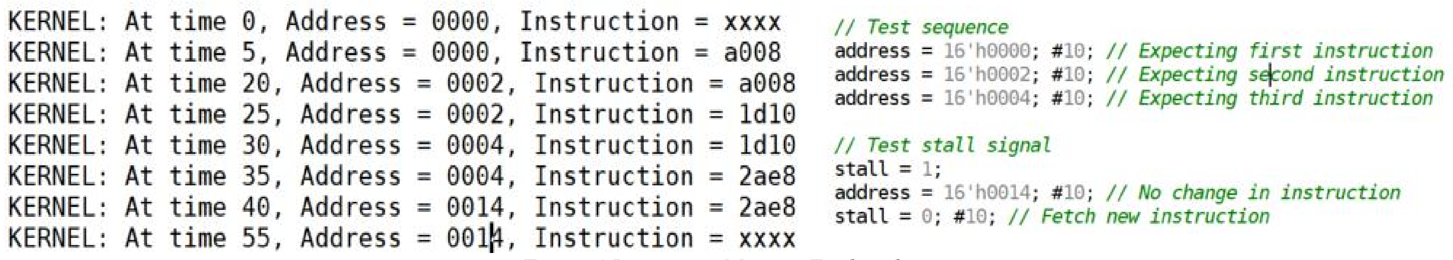
Figure 3 Control Signals Boolean Equations

By following the datapath for each instruction, and saving the control signals values in the table. We concluded the boolean equations for the control signals. In the control unit, we implemented those control signals which sends the signals based on the instruction we are working on.

Individual Components

Instruction Memory

The instruction memory module is essential for storing and fetching instructions based on the provided address. It ensures efficient execution within the pipelined architecture by supporting instruction retrieval and handling pipeline stalls. By doing so, it maintains a smooth flow of instructions through the fetch stage, crucial for the processor's overall performance and effective operation.



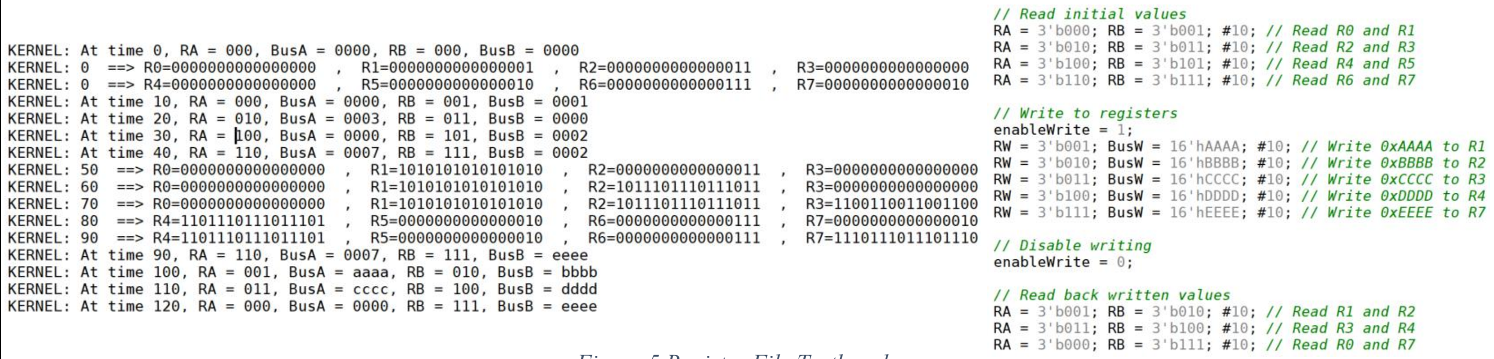
```
KERNEL: At time 0, Address = 0000, Instruction = xxxx
KERNEL: At time 5, Address = 0000, Instruction = a008
KERNEL: At time 20, Address = 0002, Instruction = a008
KERNEL: At time 25, Address = 0002, Instruction = 1d10
KERNEL: At time 30, Address = 0004, Instruction = 1d10
KERNEL: At time 35, Address = 0004, Instruction = 2ae8
KERNEL: At time 40, Address = 0014, Instruction = 2ae8
KERNEL: At time 55, Address = 0014, Instruction = xxxx
```

```
// Test sequence
address = 16'h0000; #10; // Expecting first instruction
address = 16'h0002; #10; // Expecting second instruction
address = 16'h0004; #10; // Expecting third instruction

// Test stall signal
stall = 1;
address = 16'h0014; #10; // No change in instruction
stall = 0; #10; // Fetch new instruction
```

Figure 4 Instruction Memory Testbench

Register File

The register file module is crucial for the processor's operation, providing read and write access to the general-purpose registers. It supports simultaneous reading from two registers and conditional writing to a third, ensuring efficient data handling within the pipeline. This module is vital for executing instructions and maintaining the processor's state, contributing to the overall functionality and performance of the RISC processor.



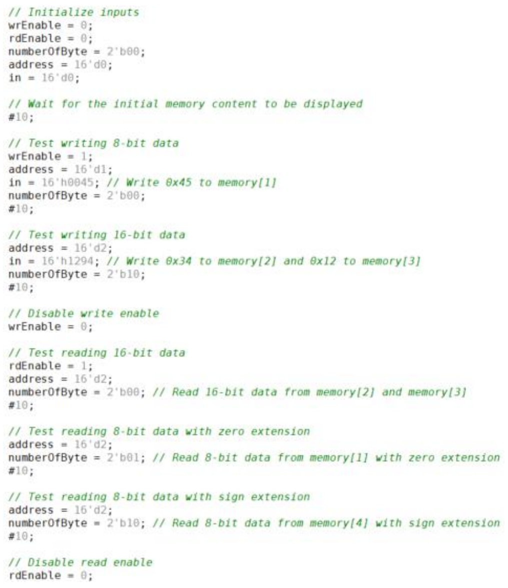
```
KERNEL: At time 0, RA = 000, BusA = 0000, RB = 000, BusB = 0000
KERNEL: 0  ==> R0=0000000000000000  ,  R1=0000000000000001  ,  R2=0000000000000011  ,  R3=0000000000000000
KERNEL: 0  ==> R4=0000000000000000  ,  R5=0000000000000010  ,  R6=0000000000000111  ,  R7=0000000000000010
KERNEL: At time 10, RA = 000, BusA = 0000, RB = 001, BusB = 0001
KERNEL: At time 20, RA = 010, BusA = 0003, RB = 011, BusB = 0000
KERNEL: At time 30, RA = 100, BusA = 0000, RB = 101, BusB = 0002
KERNEL: At time 40, RA = 110, BusA = 0007, RB = 111, BusB = 0002
KERNEL: 50  ==> R0=0000000000000000  ,  R1=1010101010101010  ,  R2=0000000000000011  ,  R3=0000000000000000
KERNEL: 60  ==> R0=0000000000000000  ,  R1=1010101010101010  ,  R2=1011101110111011  ,  R3=0000000000000000
KERNEL: 70  ==> R0=0000000000000000  ,  R1=1010101010101010  ,  R2=1011101110111011  ,  R3=1100110011001100
KERNEL: 80  ==> R4=1101110111011101  ,  R5=0000000000000010  ,  R6=0000000000000111  ,  R7=0000000000000010
KERNEL: 90  ==> R4=1101110111011101  ,  R5=0000000000000010  ,  R6=0000000000000111  ,  R7=1110111011101110
KERNEL: At time 90, RA = 110, BusA = 0007, RB = 111, BusB = eeee
KERNEL: At time 100, RA = 001, BusA = aaaa, RB = 010, BusB = bbbb
KERNEL: At time 110, RA = 011, BusA = cccc, RB = 100, BusB = dddd
KERNEL: At time 120, RA = 000, BusA = 0000, RB = 111, BusB = eeee
```

```
// Read initial values
RA = 3'b000; RB = 3'b001; #10; // Read R0 and R1
RA = 3'b010; RB = 3'b011; #10; // Read R2 and R3
RA = 3'b100; RB = 3'b101; #10; // Read R4 and R5
RA = 3'b110; RB = 3'b111; #10; // Read R6 and R7

// Write to registers
enableWrite = 1;
RW = 3'b001; BusW = 16'hAAAA; #10; // Write 0xAAAA to R1
RW = 3'b010; BusW = 16'hBBBB; #10; // Write 0xBBBB to R2
RW = 3'b011; BusW = 16'hCCCC; #10; // Write 0xCCCC to R3
RW = 3'b100; BusW = 16'hDDDD; #10; // Write 0xDDDD to R4
RW = 3'b111; BusW = 16'hEEEE; #10; // Write 0xEEEE to R7

// Disable writing
enableWrite = 0;

// Read back written values
RA = 3'b001; RB = 3'b010; #10; // Read R1 and R2
RA = 3'b011; RB = 3'b100; #10; // Read R3 and R4
RA = 3'b000; RB = 3'b111; #10; // Read R0 and R7
```

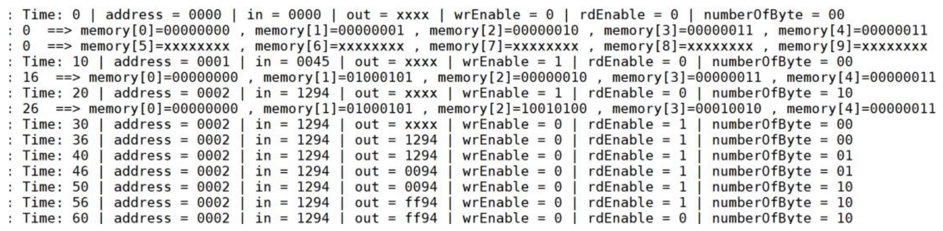
Figure 5 Register File Testbench

Data Memory

The data memory module is essential for storing and retrieving data within the processor. It supports read and write operations for both 8-bit and 16-bit data, with options for zero and sign extension. This module is crucial for efficient data handling in the pipeline, ensuring accurate and flexible memory access necessary for various instruction executions.



```
// Initialize inputs
wrEnable = 0;
rdEnable = 0;
numberOfByte = 2'b00;
address = 16'd0;
in = 16'd0;

// Wait for the initial memory content to be displayed
#10;

// Test writing 8-bit data
wrEnable = 1;
address = 16'd1;
in = 16'h0045; // Write 0x45 to memory[1]
numberOfByte = 2'b00;
#10;

// Test writing 16-bit data
address = 16'd2;
in = 16'h1294; // Write 0x34 to memory[2] and 0x12 to memory[3]
numberOfByte = 2'b10;
#10;

// Disable write enable
wrEnable = 0;

// Test reading 16-bit data
rdEnable = 1;
address = 16'd2;
numberOfByte = 2'b00; // Read 16-bit data from memory[2] and memory[3]
#10;

// Test reading 8-bit data with zero extension
address = 16'd2;
numberOfByte = 2'b01; // Read 8-bit data from memory[1] with zero extension
#10;

// Test reading 8-bit data with sign extension
address = 16'd2;
numberOfByte = 2'b10; // Read 8-bit data from memory[4] with sign extension
#10;

// Disable read enable
rdEnable = 0;
```

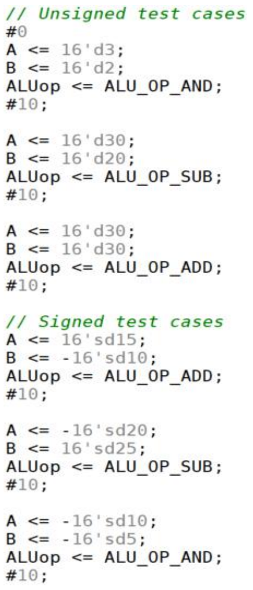


```
: Time: 0 | address = 0000 | in = 0000 | out = xxxx | wrEnable = 0 | rdEnable = 0 | numberOfByte = 00
: 0  ==> memory[0]=00000000 , memory[1]=00000001 , memory[2]=00000010 , memory[3]=00000011 , memory[4]=00000011
: 0  ==> memory[5]=xxxxxxxx , memory[6]=xxxxxxxx , memory[7]=xxxxxxxx , memory[8]=xxxxxxxx , memory[9]=xxxxxxxx
: Time: 10 | address = 0001 | in = 0045 | out = xxxx | wrEnable = 1 | rdEnable = 0 | numberOfByte = 00
: 16  ==> memory[0]=00000000 , memory[1]=01000101 , memory[2]=00000010 , memory[3]=00000011 , memory[4]=00000011
: Time: 20 | address = 0002 | in = 1294 | out = xxxx | wrEnable = 1 | rdEnable = 0 | numberOfByte = 10
: 26  ==> memory[0]=00000000 , memory[1]=01000101 , memory[2]=10010100 , memory[3]=00010010 , memory[4]=00000011
: Time: 30 | address = 0002 | in = 1294 | out = xxxx | wrEnable = 0 | rdEnable = 1 | numberOfByte = 00
: Time: 36 | address = 0002 | in = 1294 | out = 1294 | wrEnable = 0 | rdEnable = 1 | numberOfByte = 00
: Time: 40 | address = 0002 | in = 1294 | out = 1294 | wrEnable = 0 | rdEnable = 1 | numberOfByte = 01
: Time: 46 | address = 0002 | in = 1294 | out = 0094 | wrEnable = 0 | rdEnable = 1 | numberOfByte = 01
: Time: 50 | address = 0002 | in = 1294 | out = 0094 | wrEnable = 0 | rdEnable = 1 | numberOfByte = 10
: Time: 56 | address = 0002 | in = 1294 | out = ff94 | wrEnable = 0 | rdEnable = 1 | numberOfByte = 10
: Time: 60 | address = 0002 | in = 1294 | out = ff94 | wrEnable = 0 | rdEnable = 0 | numberOfByte = 10
```

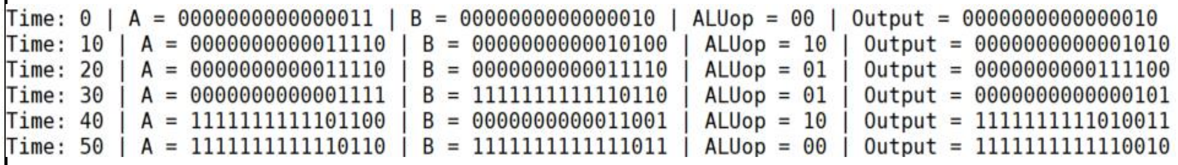
Figure 6 Data Memory Testbench

ALU

The ALU (Arithmetic Logic Unit) module is fundamental for performing arithmetic and logical operations within the processor. It handles addition, subtraction, and bitwise AND operations based on the ALUop control signal. This module is crucial for executing various instructions efficiently, enabling the processor to perform essential computations and logic operations required by the instruction set architecture (ISA).



```
// Unsigned test cases
#0
A <= 16'd3;
B <= 16'd2;
ALUop <= ALU_OP_AND;
#10;

A <= 16'd30;
B <= 16'd20;
ALUop <= ALU_OP_SUB;
#10;

A <= 16'd30;
B <= 16'd30;
ALUop <= ALU_OP_ADD;
#10;

// Signed test cases
A <= 16'sd15;
B <= -16'sd10;
ALUop <= ALU_OP_ADD;
#10;

A <= -16'sd20;
B <= 16'sd25;
ALUop <= ALU_OP_SUB;
#10;

A <= -16'sd10;
B <= -16'sd5;
ALUop <= ALU_OP_AND;
#10;
```

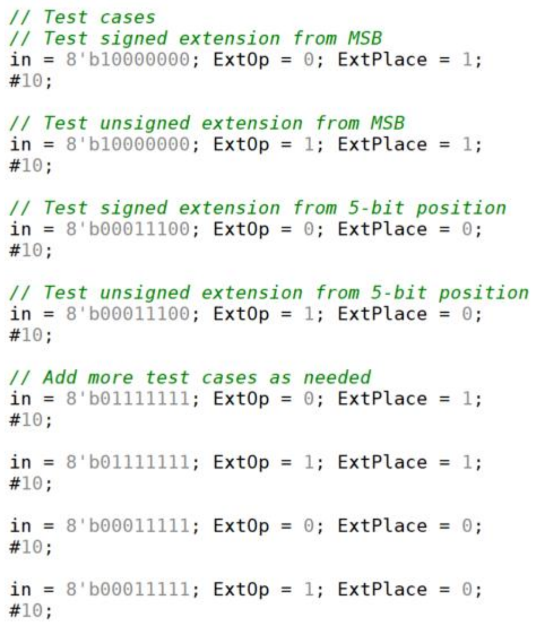


```
Time: 0 | A = 0000000000000011 | B = 0000000000000010 | ALUop = 00 | Output = 0000000000000010
Time: 10 | A = 0000000000011110 | B = 0000000000010100 | ALUop = 10 | Output = 0000000000001010
Time: 20 | A = 0000000000011110 | B = 0000000000011110 | ALUop = 01 | Output = 0000000000111100
Time: 30 | A = 0000000000001111 | B = 1111111111110110 | ALUop = 01 | Output = 0000000000000101
Time: 40 | A = 1111111111101100 | B = 0000000000011001 | ALUop = 10 | Output = 1111111111010011
Time: 50 | A = 1111111111110110 | B = 1111111111111011 | ALUop = 00 | Output = 1111111111110010
```

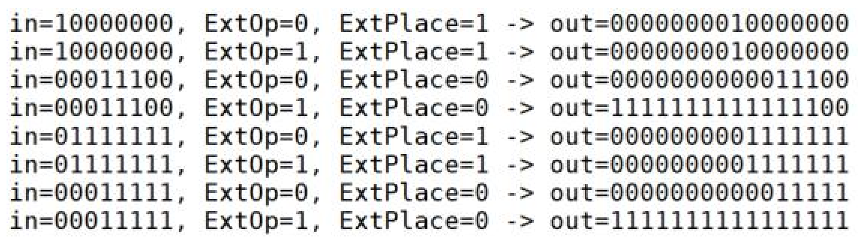
Figure 7 ALU Testbench

Extender

The extender module is essential for adjusting the bit-width of data within the processor, supporting both signed and unsigned extensions. It can extend 8-bit inputs to 16-bit outputs, either by zero extension or sign extension, depending on the control signals. This module ensures that immediate values and offsets are correctly processed, which is crucial for accurate instruction execution and overall processor functionality.



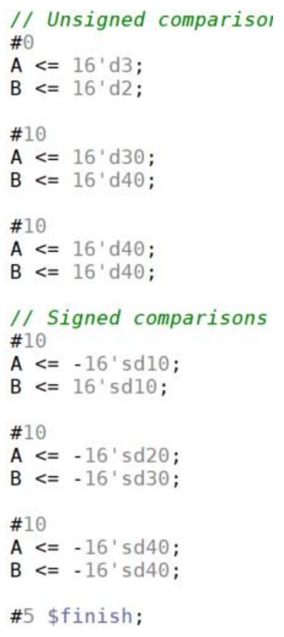
```
// Test cases
// Test signed extension from MSB
in = 8'b10000000; ExtOp = 0; ExtPlace = 1;
#10;

// Test unsigned extension from MSB
in = 8'b10000000; ExtOp = 1; ExtPlace = 1;
#10;

// Test signed extension from 5-bit position
in = 8'b00011100; ExtOp = 0; ExtPlace = 0;
#10;

// Test unsigned extension from 5-bit position
in = 8'b00011100; ExtOp = 1; ExtPlace = 0;
#10;

// Add more test cases as needed
in = 8'b01111111; ExtOp = 0; ExtPlace = 1;
#10;

in = 8'b01111111; ExtOp = 1; ExtPlace = 1;
#10;

in = 8'b00011111; ExtOp = 0; ExtPlace = 0;
#10;

in = 8'b00011111; ExtOp = 1; ExtPlace = 0;
#10;
```



```
in=10000000, ExtOp=0, ExtPlace=1 -> out=0000000010000000
in=10000000, ExtOp=1, ExtPlace=1 -> out=0000000010000000
in=00011100, ExtOp=0, ExtPlace=0 -> out=0000000000011100
in=00011100, ExtOp=1, ExtPlace=0 -> out=1111111111111100
in=01111111, ExtOp=0, ExtPlace=1 -> out=0000000001111111
in=01111111, ExtOp=1, ExtPlace=1 -> out=0000000001111111
in=00011111, ExtOp=0, ExtPlace=0 -> out=0000000000011111
in=00011111, ExtOp=1, ExtPlace=0 -> out=1111111111111111
```

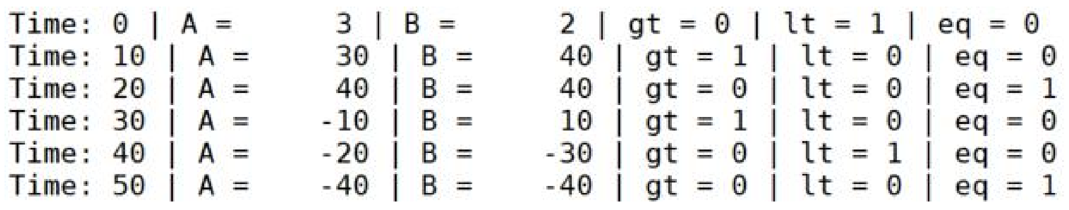
Figure 8 Extender Testbench

Compare



```
// Unsigned compariso
#0
A <= 16'd3;
B <= 16'd2;

#10
A <= 16'd30;
B <= 16'd40;

#10
A <= 16'd40;
B <= 16'd40;

// Signed comparisons
#10
A <= -16'sd10;
B <= 16'sd10;

#10
A <= -16'sd20;
B <= -16'sd30;

#10
A <= -16'sd40;
B <= -16'sd40;

#5 $finish;
```

The compare module is a critical component for performing comparisons between two 16-bit signed inputs. It determines whether one value is greater than, less than, or equal to the other, setting the corresponding output signals. This functionality is essential for executing conditional branch instructions and other operations that rely on value comparisons, ensuring accurate decision-making within the processor.



```
Time: 0 | A =      3 | B =      2 | gt = 0 | lt = 1 | eq = 0
Time: 10 | A =     30 | B =     40 | gt = 1 | lt = 0 | eq = 0
Time: 20 | A =     40 | B =     40 | gt = 0 | lt = 0 | eq = 1
Time: 30 | A =    -10 | B =     10 | gt = 1 | lt = 0 | eq = 0
Time: 40 | A =    -20 | B =    -30 | gt = 0 | lt = 1 | eq = 0
Time: 50 | A =    -40 | B =    -40 | gt = 0 | lt = 0 | eq = 1
```

Figure 9 Compare Testbench

Control Unit

The ControlUnit.v is a critical module in the pipelined RISC processor, responsible for generating and managing the control signals that dictate the operation of various components within the processor. This module ensures that each stage of the pipeline operates cohesively, executing instructions accurately and efficiently. It includes three submodules: MainAluControl, which generates control signals for arithmetic, logic, memory, and control flow operations based on the opcode; PcControl, which manages the program counter by determining the source of the next PC value and handling branch, jump, and return instructions; and HazardDetect, which resolves data hazards by generating stall signals and forwarding paths to maintain data integrity across the pipeline.

The integration of these submodules within ControlUnit.v allows for a robust and efficient control mechanism, ensuring that the processor can handle complex instruction sequences without errors. The MainAluControl module provides the necessary signals to execute a wide range of instructions, while the PcControl module ensures that the instruction flow follows the correct path, particularly during conditional and unconditional branches. Meanwhile, the HazardDetect module prevents pipeline hazards by stalling the pipeline or forwarding data as needed, thus maintaining smooth and accurate instruction execution. Together, these components enable the processor to achieve high performance and reliability.

Main ALU Control

The MainAluControl module generates essential control signals for the ALU and other components based on the instruction's opcode, destination register, and mode. It outputs a 17-bit control signal array that dictates the behavior of various processor components during instruction execution. This module ensures accurate operation of arithmetic, logic, memory, and control flow instructions, facilitating correct and efficient execution of the processor's instruction set, thereby enabling seamless instruction handling and performance optimization.

PC Control

The PcControl module determines the source of the next program counter (PC) value, managing control flow based on the current opcode and comparison results (GT, LT, EQ). It sets the PcSrc and kill signals to handle branches, jumps, and returns effectively. This module is crucial for directing the execution flow, ensuring the processor follows the correct sequence of instructions and accurately processes conditional branches and jumps, thus maintaining the integrity of the instruction flow.

Hazard Detect

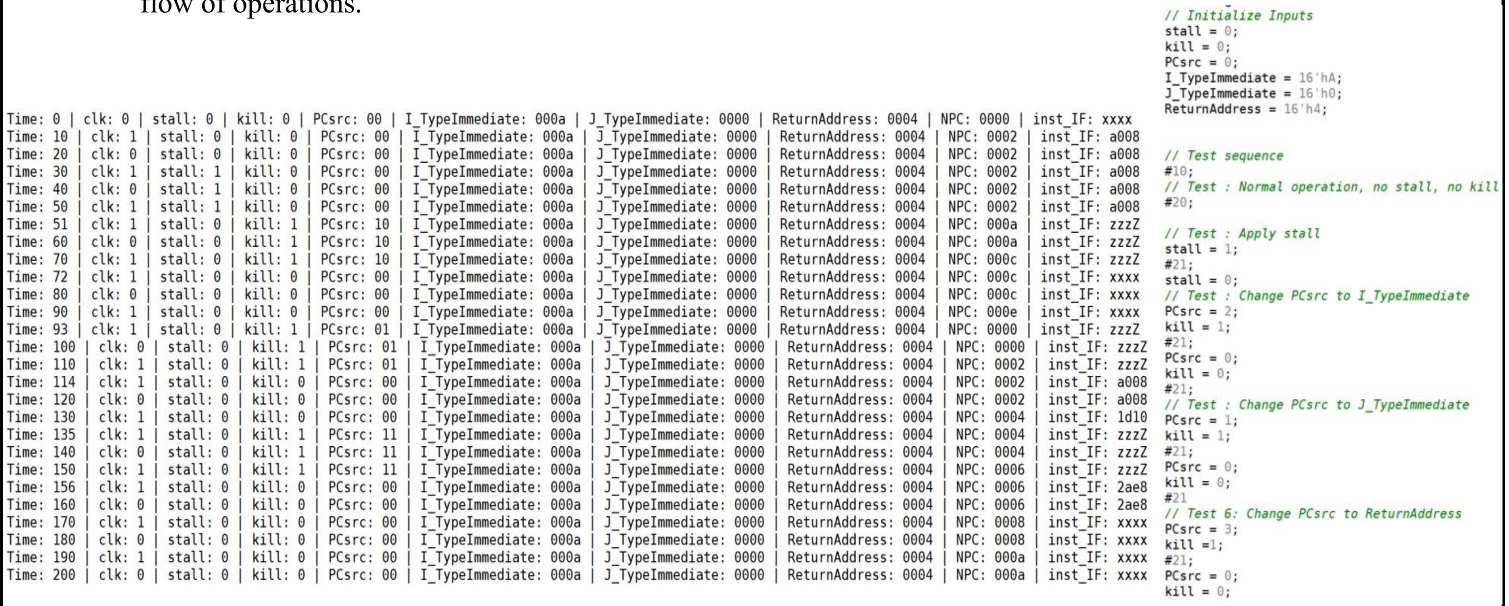
The HazardDetect module identifies and resolves data hazards within the pipeline. It detects read-after-write (RAW) hazards and generates stall signals to pause the pipeline when necessary. Additionally, it provides forwarding paths (ForwardA and ForwardB) to resolve data dependencies by bypassing data from later stages back to earlier ones. This module is essential for maintaining correct data flow and preventing execution errors due to hazards, ensuring smooth and reliable pipeline operation.

Stages

Our processor design is split into four stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), and Memory Access (MEM). Each stage performs distinct tasks, enabling parallel processing and improved efficiency. The IF stage retrieves instructions from memory, the ID stage decodes instructions and reads registers, the EXE stage performs arithmetic and logical operations, and the MEM stage handles data memory access. Splitting these tasks into stages allows for a pipelined architecture, increasing instruction throughput and overall performance.

Instruction Fetch Stage

The Instruction Fetch (IF) stage is responsible for retrieving the next instruction to be executed from the instruction memory. It uses the program counter (PC) to address the instruction memory and handles pipeline control signals like stall and kill. The fetched instruction is then forwarded to the next stage. This stage ensures that instructions are continuously fed into the pipeline, maintaining a smooth flow of operations.



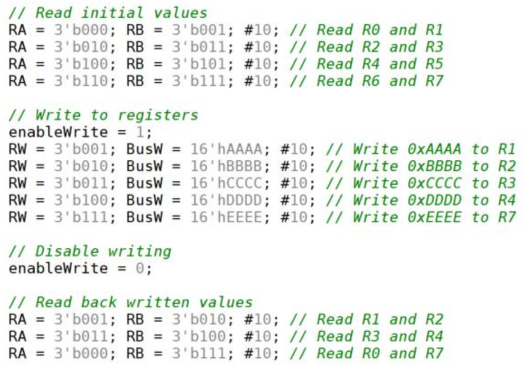
```
Time: 0 | clk: 0 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0000 | inst_IF: xxxx
Time: 10 | clk: 1 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0002 | inst_IF: a008
Time: 20 | clk: 0 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0002 | inst_IF: a008
Time: 30 | clk: 1 | stall: 1 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0002 | inst_IF: a008
Time: 40 | clk: 0 | stall: 1 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0002 | inst_IF: a008
Time: 50 | clk: 1 | stall: 1 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0002 | inst_IF: a008
Time: 51 | clk: 1 | stall: 0 | kill: 1 | PCsrc: 10 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 000a | inst_IF: zzzZ
Time: 60 | clk: 0 | stall: 0 | kill: 1 | PCsrc: 10 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 000a | inst_IF: zzzZ
Time: 70 | clk: 1 | stall: 0 | kill: 1 | PCsrc: 10 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 000c | inst_IF: zzzZ
Time: 72 | clk: 1 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 000c | inst_IF: xxxx
Time: 80 | clk: 0 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 000c | inst_IF: xxxx
Time: 90 | clk: 1 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 000e | inst_IF: xxxx
Time: 93 | clk: 1 | stall: 0 | kill: 1 | PCsrc: 01 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0000 | inst_IF: zzzZ
Time: 100 | clk: 0 | stall: 0 | kill: 1 | PCsrc: 01 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0000 | inst_IF: zzzZ
Time: 110 | clk: 1 | stall: 0 | kill: 1 | PCsrc: 01 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0002 | inst_IF: zzzZ
Time: 114 | clk: 1 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0002 | inst_IF: a008
Time: 120 | clk: 0 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0002 | inst_IF: a008
Time: 130 | clk: 1 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0004 | inst_IF: 1d10
Time: 135 | clk: 1 | stall: 0 | kill: 1 | PCsrc: 11 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0004 | inst_IF: zzzZ
Time: 140 | clk: 0 | stall: 0 | kill: 1 | PCsrc: 11 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0004 | inst_IF: zzzZ
Time: 150 | clk: 1 | stall: 0 | kill: 1 | PCsrc: 11 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0006 | inst_IF: zzzZ
Time: 156 | clk: 1 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0006 | inst_IF: 2ae8
Time: 160 | clk: 0 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0006 | inst_IF: 2ae8
Time: 170 | clk: 1 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0008 | inst_IF: xxxx
Time: 180 | clk: 0 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 0008 | inst_IF: xxxx
Time: 190 | clk: 1 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 000a | inst_IF: xxxx
Time: 200 | clk: 0 | stall: 0 | kill: 0 | PCsrc: 00 | I_TypeImmediate: 000a | J_TypeImmediate: 0000 | ReturnAddress: 0004 | NPC: 000a | inst_IF: xxxx
```

```
// Initialize Inputs
stall = 0;
kill = 0;
PCsrc = 0;
I_TypeImmediate = 16'hA;
J_TypeImmediate = 16'h0;
ReturnAddress = 16'h4;


// Test sequence
#10;
// Test : Normal operation, no stall, no kill
#20;

// Test : Apply stall
stall = 1;
#21;
stall = 0;
// Test : Change PCsrc to I_TypeImmediate
PCsrc = 2;
kill = 1;
#21;
PCsrc = 0;
kill = 0;
#21;
// Test : Change PCsrc to J_TypeImmediate
PCsrc = 1;
kill = 1;
#21;
PCsrc = 0;
kill = 0;
#21
// Test 6: Change PCsrc to ReturnAddress
PCsrc = 3;
kill =1;
#21;
PCsrc = 0;
kill = 0;
```

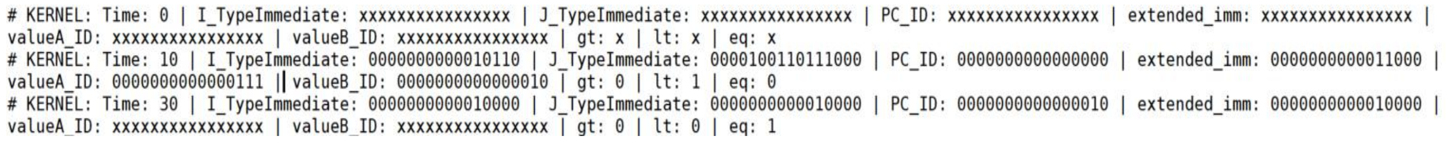
Figure 10 Instruction Fetch Stage Testbench

Instruction Decode Stage

The Instruction Decode (ID) stage decodes the fetched instruction and reads the necessary data from the register file. It identifies the instruction type and extracts operands and immediate values. This stage also handles forwarding and branching decisions, ensuring that data dependencies are resolved and branch instructions are correctly processed. The decoded instruction and read data are then passed to the Execute stage.



```
// Read initial values
RA = 3'b000; RB = 3'b001; #10; // Read R0 and R1
RA = 3'b010; RB = 3'b011; #10; // Read R2 and R3
RA = 3'b100; RB = 3'b101; #10; // Read R4 and R5
RA = 3'b110; RB = 3'b111; #10; // Read R6 and R7

// Write to registers
enableWrite = 1;
RW = 3'b001; BusW = 16'hAAAA; #10; // Write 0xAAAA to R1
RW = 3'b010; BusW = 16'hBBBB; #10; // Write 0xBBBB to R2
RW = 3'b011; BusW = 16'hCCCC; #10; // Write 0xCCCC to R3
RW = 3'b100; BusW = 16'hDDDD; #10; // Write 0xDDDD to R4
RW = 3'b111; BusW = 16'hEEEE; #10; // Write 0xEEEE to R7

// Disable writing
enableWrite = 0;

// Read back written values
RA = 3'b001; RB = 3'b010; #10; // Read R1 and R2
RA = 3'b011; RB = 3'b100; #10; // Read R3 and R4
RA = 3'b000; RB = 3'b111; #10; // Read R0 and R7
```

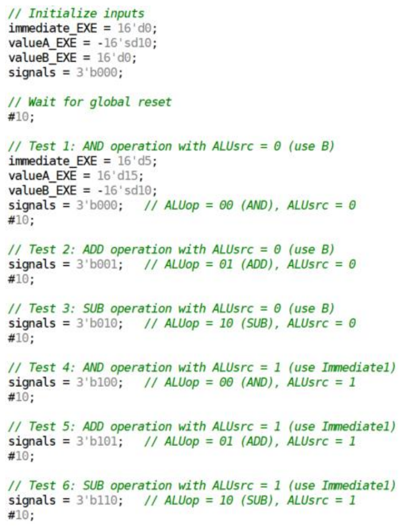


```
# KERNEL: Time: 0 | I_TypeImmediate: xxxxxxxxxxxxxxxx | J_TypeImmediate: xxxxxxxxxxxxxxxx | PC_ID: xxxxxxxxxxxxxxxx | extended_imm: xxxxxxxxxxxxxxxx |
valueA_ID: xxxxxxxxxxxxxxxx | valueB_ID: xxxxxxxxxxxxxxxx | gt: x | lt: x | eq: x
# KERNEL: Time: 10 | I_TypeImmediate: 0000000000010110 | J_TypeImmediate: 0000100110111000 | PC_ID: 0000000000000000 | extended_imm: 0000000000011000 |
valueA_ID: 0000000000000111 || valueB_ID: 0000000000000010 | gt: 0 | lt: 1 | eq: 0
# KERNEL: Time: 30 | I_TypeImmediate: 0000000000010000 | J_TypeImmediate: 0000000000010000 | PC_ID: 0000000000000010 | extended_imm: 0000000000010000 |
valueA_ID: xxxxxxxxxxxxxxxx | valueB_ID: xxxxxxxxxxxxxxxx | gt: 0 | lt: 0 | eq: 1
```

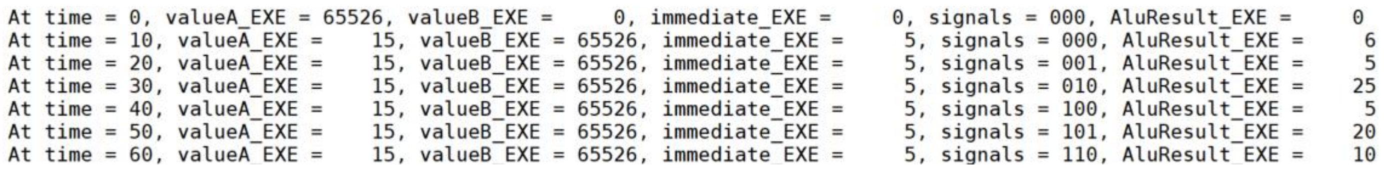
Figure 11 Instruction Decode Stage Testbench

Execution Stage

The Execute (EXE) stage performs arithmetic and logical operations specified by the instruction. It uses an Arithmetic Logic Unit (ALU) to process operands from the register file or immediate values. The results of these operations, such as ALU results and condition flags, are generated in this stage. The EXE stage is crucial for carrying out the core computational tasks of the processor.



```
// Initialize inputs
immediate_EXE = 16'd0;
valueA_EXE = -16'sd10;
valueB_EXE = 16'd0;
signals = 3'b000;

// Wait for global reset
#10;

// Test 1: AND operation with ALUsrc = 0 (use B)
immediate_EXE = 16'd5;
valueA_EXE = 16'd15;
valueB_EXE = -16'sd10;
signals = 3'b000;   // ALUop = 00 (AND), ALUsrc = 0
#10;

// Test 2: ADD operation with ALUsrc = 0 (use B)
signals = 3'b001;   // ALUop = 01 (ADD), ALUsrc = 0
#10;

// Test 3: SUB operation with ALUsrc = 0 (use B)
signals = 3'b010;   // ALUop = 10 (SUB), ALUsrc = 0
#10;

// Test 4: AND operation with ALUsrc = 1 (use Immediate1)
signals = 3'b100;   // ALUop = 00 (AND), ALUsrc = 1
#10;

// Test 5: ADD operation with ALUsrc = 1 (use Immediate1)
signals = 3'b101;   // ALUop = 01 (ADD), ALUsrc = 1
#10;

// Test 6: SUB operation with ALUsrc = 1 (use Immediate1)
signals = 3'b110;   // ALUop = 10 (SUB), ALUsrc = 1
#10;
```

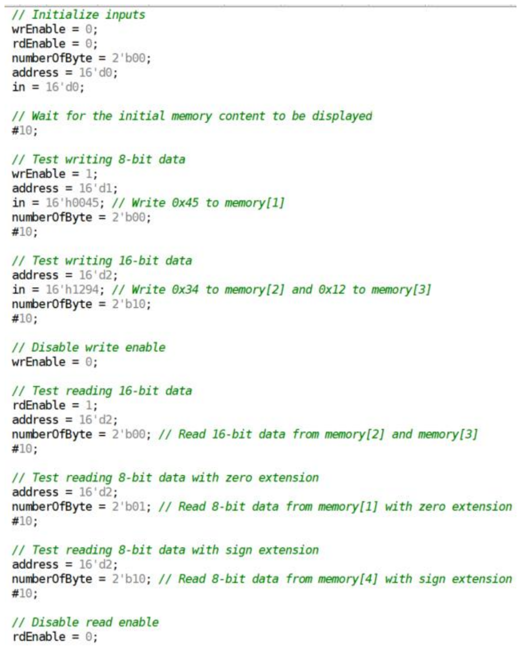


```
At time = 0, valueA_EXE = 65526, valueB_EXE =     0, immediate_EXE =     0, signals = 000, AluResult_EXE =     0
At time = 10, valueA_EXE =    15, valueB_EXE = 65526, immediate_EXE =     5, signals = 000, AluResult_EXE =     6
At time = 20, valueA_EXE =    15, valueB_EXE = 65526, immediate_EXE =     5, signals = 001, AluResult_EXE =     5
At time = 30, valueA_EXE =    15, valueB_EXE = 65526, immediate_EXE =     5, signals = 010, AluResult_EXE =    25
At time = 40, valueA_EXE =    15, valueB_EXE = 65526, immediate_EXE =     5, signals = 100, AluResult_EXE =     5
At time = 50, valueA_EXE =    15, valueB_EXE = 65526, immediate_EXE =     5, signals = 101, AluResult_EXE =    20
At time = 60, valueA_EXE =    15, valueB_EXE = 65526, immediate_EXE =     5, signals = 110, AluResult_EXE =    10
```

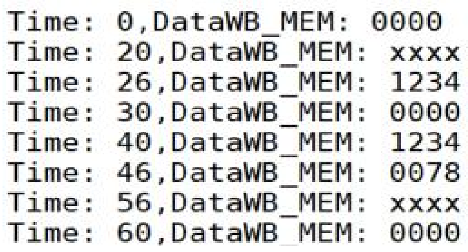
Figure 12 Memory Stage Testbench

Memory Stage

The Memory Access (MEM) stage handles the reading and writing of data to and from the data memory. Depending on the instruction type, it may perform load or store operations, accessing the memory address calculated in the Execute stage. This stage ensures that data is correctly read from or written to memory, providing necessary data for subsequent instructions or storing results from computations.



```
// Initialize inputs
wrEnable = 0;
rdEnable = 0;
numberOfByte = 2'b00;
address = 16'd0;
in = 16'd0;

// Wait for the initial memory content to be displayed
#10;

// Test writing 8-bit data
wrEnable = 1;
address = 16'd1;
in = 16'h0045; // Write 0x45 to memory[1]
numberOfByte = 2'b00;
#10;

// Test writing 16-bit data
address = 16'd2;
in = 16'h1294; // Write 0x34 to memory[2] and 0x12 to memory[3]
numberOfByte = 2'b10;
#10;

// Disable write enable
wrEnable = 0;

// Test reading 16-bit data
rdEnable = 1;
address = 16'd2;
numberOfByte = 2'b00; // Read 16-bit data from memory[2] and memory[3]
#10;

// Test reading 8-bit data with zero extension
address = 16'd2;
numberOfByte = 2'b01; // Read 8-bit data from memory[1] with zero extension
#10;

// Test reading 8-bit data with sign extension
address = 16'd2;
numberOfByte = 2'b10; // Read 8-bit data from memory[4] with sign extension
#10;

// Disable read enable
rdEnable = 0;
```



```
Time: 0,DataWB_MEM: 0000
Time: 20,DataWB_MEM: xxxx
Time: 26,DataWB_MEM: 1234
Time: 30,DataWB_MEM: 0000
Time: 40,DataWB_MEM: 1234
Time: 46,DataWB_MEM: 0078
Time: 56,DataWB_MEM: xxxx
Time: 60,DataWB_MEM: 0000
```

Figure 13 Memory Stage Testbench

Full Pipelined Processor

The PipelineProcessor module implements a pipelined RISC processor, efficiently managing the execution of instructions across multiple stages. It begins with the clock generation, ensuring synchronized operations throughout the processor. Key stages include Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), and Memory Access (MEM), each handled by specific modules. The IF stage retrieves instructions and updates the program counter, while the ID stage decodes instructions, reads registers, and prepares data for execution. The EXE stage performs arithmetic and logic operations, and the MEM stage handles data memory access, ensuring smooth data flow and execution.

The processor is governed by a comprehensive control unit consisting of MainAluControl, PcControl, and HazardDetect modules. These modules generate control signals, manage the program counter, and resolve data hazards, respectively. Pipeline registers (IF2ID, ID2EXE, EXE2MEM, MEM2WB) are used to transfer data between stages, maintaining data integrity and synchronization. This architecture enhances instruction throughput and processor performance by allowing multiple instructions to be processed concurrently in different pipeline stages.

We will have 5 programs to showcase our working processor.

TEST CASES

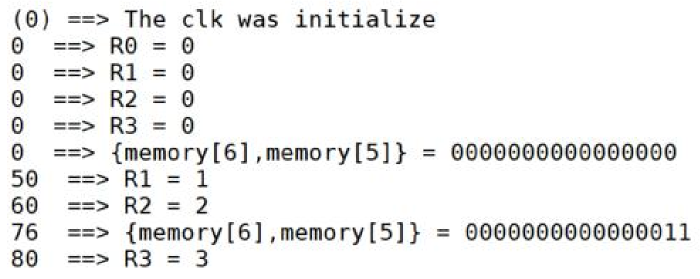
First Program: Sum of Numbers

The program will load two numbers from the data memory, add them and then store the result back in the memory.

Code

```
{instructionMemory[1],instructionMemory[0]} = {LW, R1, R0, 1'b0, 5'b00001};
{instructionMemory[3],instructionMemory[2]} = {LW, R2, R0, 1'b0, 5'b00011};
{instructionMemory[5],instructionMemory[4]} = {ADD, R3, R1, R2, 3'b000};
{instructionMemory[7],instructionMemory[6]} = {SW, R3, R0, 1'b0, 5'b00101};
```

Results and Waveforms



```
(0) ==> The clk was initialize
0  ==> R0 = 0
0  ==> R1 = 0
0  ==> R2 = 0
0  ==> R3 = 0
0  ==> {memory[6],memory[5]} = 0000000000000000
50  ==> R1 = 1
60  ==> R2 = 2
76  ==> {memory[6],memory[5]} = 0000000000000011
80  ==> R3 = 3
```

Figure 14 Program 1 Output

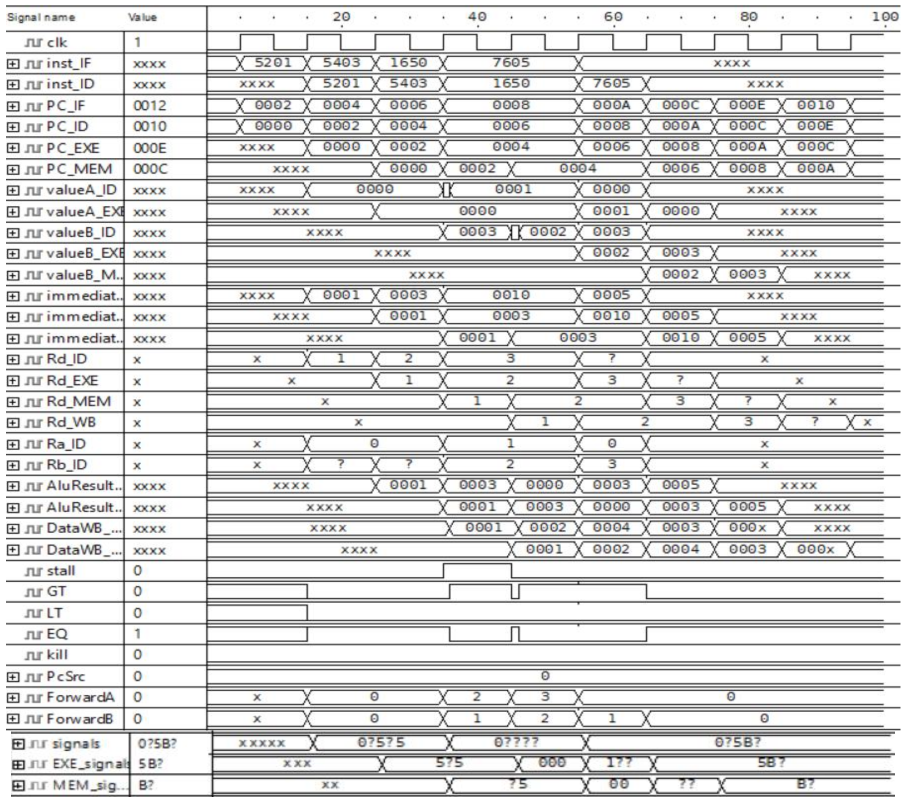


Figure 15 Program 1 waveform

Explanation

- ✧ At second 50, R1 is loaded with the value in memory[1,2] which is 1
- ✧ At second 60, R2 is loaded with the value in memory[3,4] which is 2
- ✧ At second 76, the value in R3 is stored in the memory[5,6] which is 3
- ✧ At second 80, the output of the ALU is saved in R3

Important Point

The ADD operation result is computed and available in the ALU stage before it is written back to the register file in the write-back stage.The SW instruction can access the result from the ALU stage (due to forwarding) and store it into memory before the ADD instruction completes the write-back to R3.This ensures that the correct value is written into memory without waiting for the register write-back, optimizing the execution time and reducing stalls in the pipeline. This can be seen by the time the value was stored in the memory (76ns) and the time it was saved in the register file (80ns).

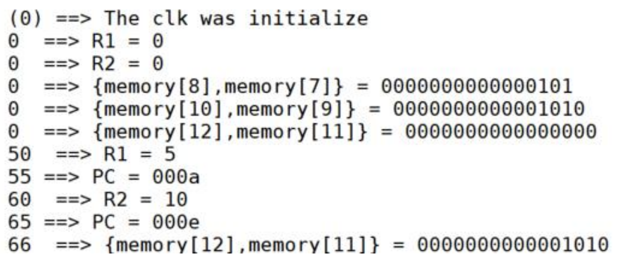
Second Program: Maximum of Two Numbers using Conditional Branch

The program will load two values from the memory and store the maximum of the two numbers back in the memory.

Code

```
{instructionMemory[1],instructionMemory[0]} = {LW, R1, R0, 1'b0, 5'b00111};
{instructionMemory[3],instructionMemory[2]} = {LW, R2, R0, 1'b0, 5'b01001};
{instructionMemory[5],instructionMemory[4]} = {BGT, R1, R2, 5'b00100};
{instructionMemory[7],instructionMemory[6]} = {SW, R2, R0, 1'b0, 5'b01011};
{instructionMemory[9],instructionMemory[8]} = {JMP, 12'h00E};
{instructionMemory[11],instructionMemory[10]} = {SW, R1, R0, 1'b0, 5'b01011};
```

Results and Waveforms



```
(0) ==> The clk was initialize
0  ==> R1 = 0
0  ==> R2 = 0
0  ==> {memory[8],memory[7]} = 0000000000000101
0  ==> {memory[10],memory[9]} = 0000000000001010
0  ==> {memory[12],memory[11]} = 0000000000000000
50  ==> R1 = 5
55 ==> PC = 000a
60  ==> R2 = 10
65 ==> PC = 000e
66  ==> {memory[12],memory[11]} = 0000000000001010
```

Figure 16 Program 2 Output

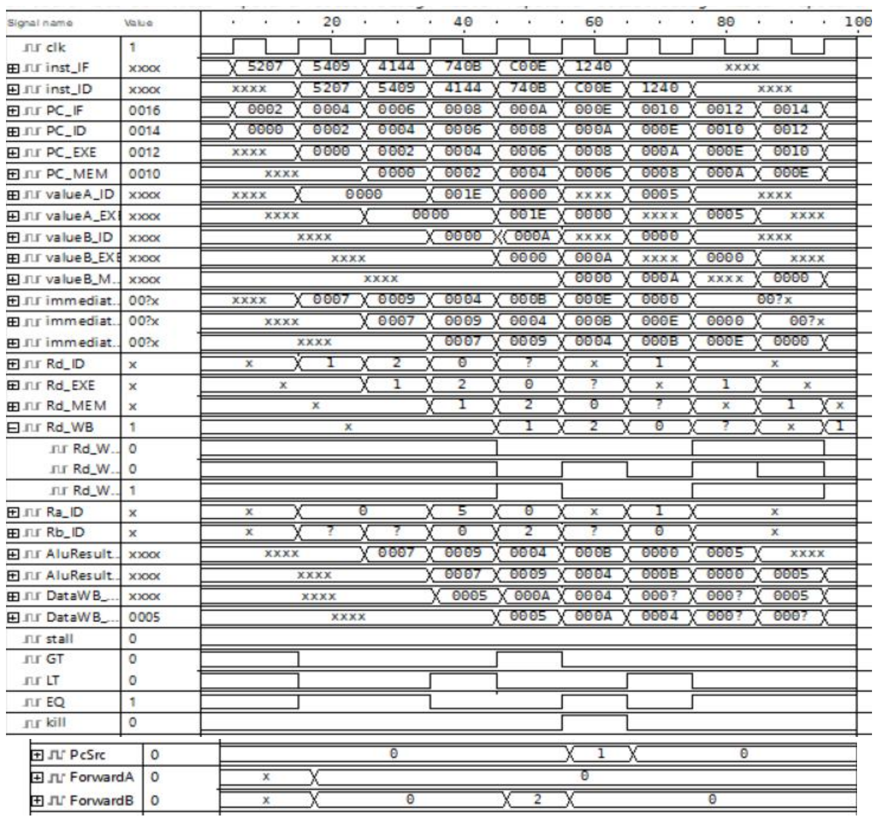


Figure 17 Program 2 Waveform

Explanation

- ✧ At second 50, R1 is loaded with the value in memory[7,8] which is 5
- ✧ At second 55, we can notice that the PC is A
- ✧ At second 60, R2 is loaded with the value in memory[9,10] which is 10
- ✧ At second 65, we can notice that the branch was taken and the PC is now E
- ✧ At second 66, the value in R2 which is the maximum is stored in the memory[11,12] which is 10

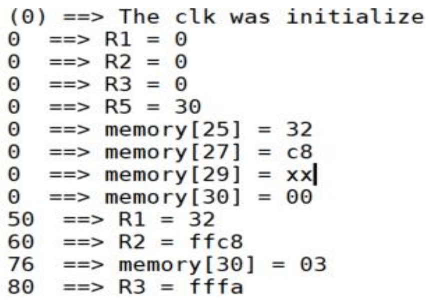
Third Program: Load, Extend, Add and Store

The program will load two byte from two memory cells one with a zero extension and one with a sign extension, add them in a register. Then store a byte (an immediate value) in the memory.

Code

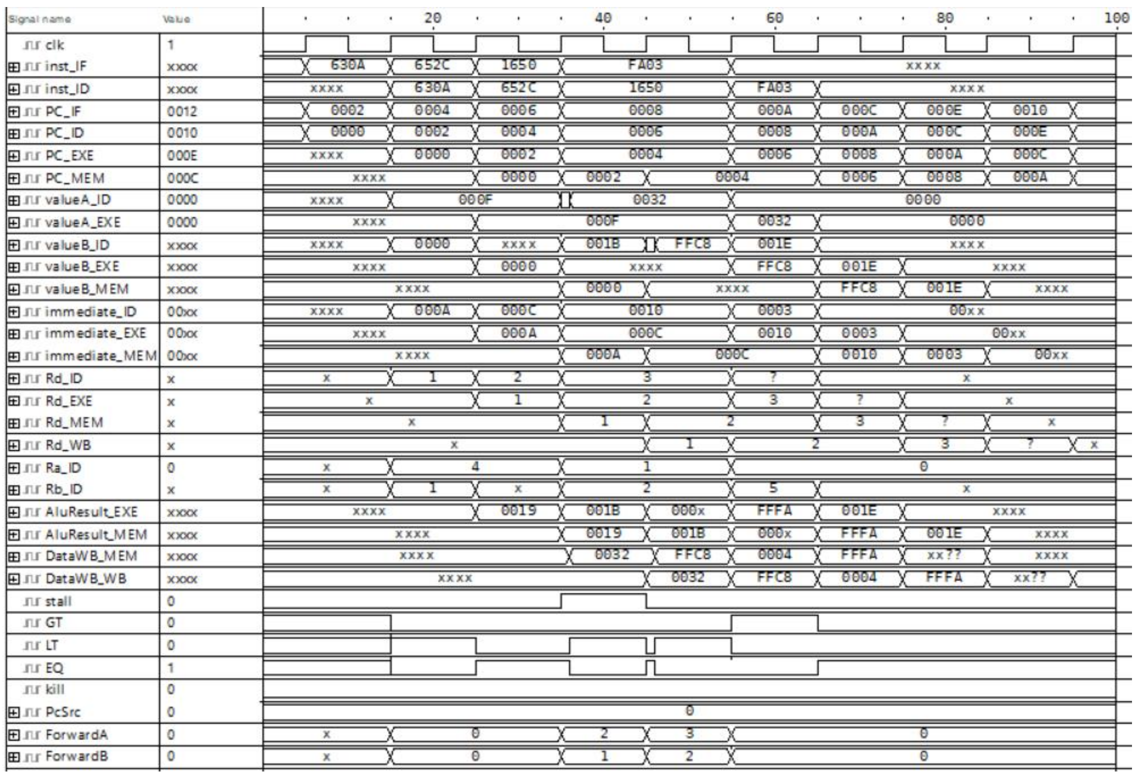
```
{instructionMemory[1],instructionMemory[0]} = {LBu, R1, R4, 1'b0, 5'b01010};

{instructionMemory[3],instructionMemory[2]} = {LBs, R2, R4, 1'b1, 5'b01100};

{instructionMemory[5],instructionMemory[4]} = {ADD, R3, R1, R2, 3'b000}

{instructionMemory[7],instructionMemory[6]} = {SV, R5, 1'b0, 8'b000000011};
```

Result



```
(0) ==> The clk was initialize
0  ==> R1 = 0
0  ==> R2 = 0
0  ==> R3 = 0
0  ==> R5 = 30
0  ==> memory[25] = 32
0  ==> memory[27] = c8
0  ==> memory[29] = xx
0  ==> memory[30] = 00
50  ==> R1 = 32
60  ==> R2 = ffc8
76  ==> memory[30] = 03
80  ==> R3 = fffa
```

Figure 18 Program 3 Output



Signal name	Value
clk	1
inst_IF	xxxx
inst_ID	xxxx
PC_IF	0012
PC_ID	0010
PC_EXE	000E
PC_MEM	000C
valueA_ID	0000
valueA_EXE	0000
valueB_ID	xxxx
valueB_EXE	xxxx
valueB_MEM	xxxx
immediate_ID	00xx
immediate_EXE	00xx
immediate_MEM	00xx
Rd_ID	x
Rd_EXE	x
Rd_MEM	x
Rd_WB	x
Ra_ID	0
Rb_ID	x
AluResult_EXE	xxxx
AluResult_MEM	xxxx
DataWB_MEM	xxxx
DataWB_WB	xxxx
stall	0
GT	0
LT	0
EQ	1
kill	0
PcSrc	0
ForwardA	0
ForwardB	0

Figure 19 Program 3 Waveform

Explanation

- ✧ At second 50, the value in memory[25] is loaded on R1 with a zero extension
- ✧ At second 60, the value in memory[27] is loaded on R2 with a sign extension
- ✧ At second 76, the value 3 is stored in memory[3].
- ✧ At second 80, the alu result adding R1 and R2 is written on R3.

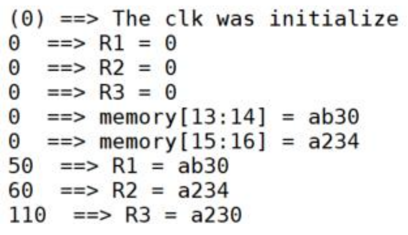
Fourth Program: Compare and Conditional Branch with AND

The program will load two values from the memory, call a subroutine that will do and AND operation if R2 is less than R1. Then return to the main program.

Code

```
{instructionMemory[1], instructionMemory[0]} = {LW, R1, R0, 1'b0, 5'b01101};
{instructionMemory[3], instructionMemory[2]} = {LW, R2, R0, 1'b0, 5'b01111};
{instructionMemory[5], instructionMemory[4]} = {CALL, 12'h00A};
{instructionMemory[11],instructionMemory[10]} = {BLT, R2, R1,1'b0,5'd2};
{instructionMemory[13],instructionMemory[12]} = {JMP, 12'd16};
{instructionMemory[15],instructionMemory[14]} = {AND, R3, R1, R2, 3'b000};
{instructionMemory[17],instructionMemory[16]} = {RET, 12'b0};
```

Result



```
(0) ==> The clk was initialize
0  ==> R1 = 0
0  ==> R2 = 0
0  ==> R3 = 0
0  ==> memory[13:14] = ab30
0  ==> memory[15:16] = a234
50  ==> R1 = ab30
60  ==> R2 = a234
110  ==> R3 = a230
```

Figure 20 Program 4 Output

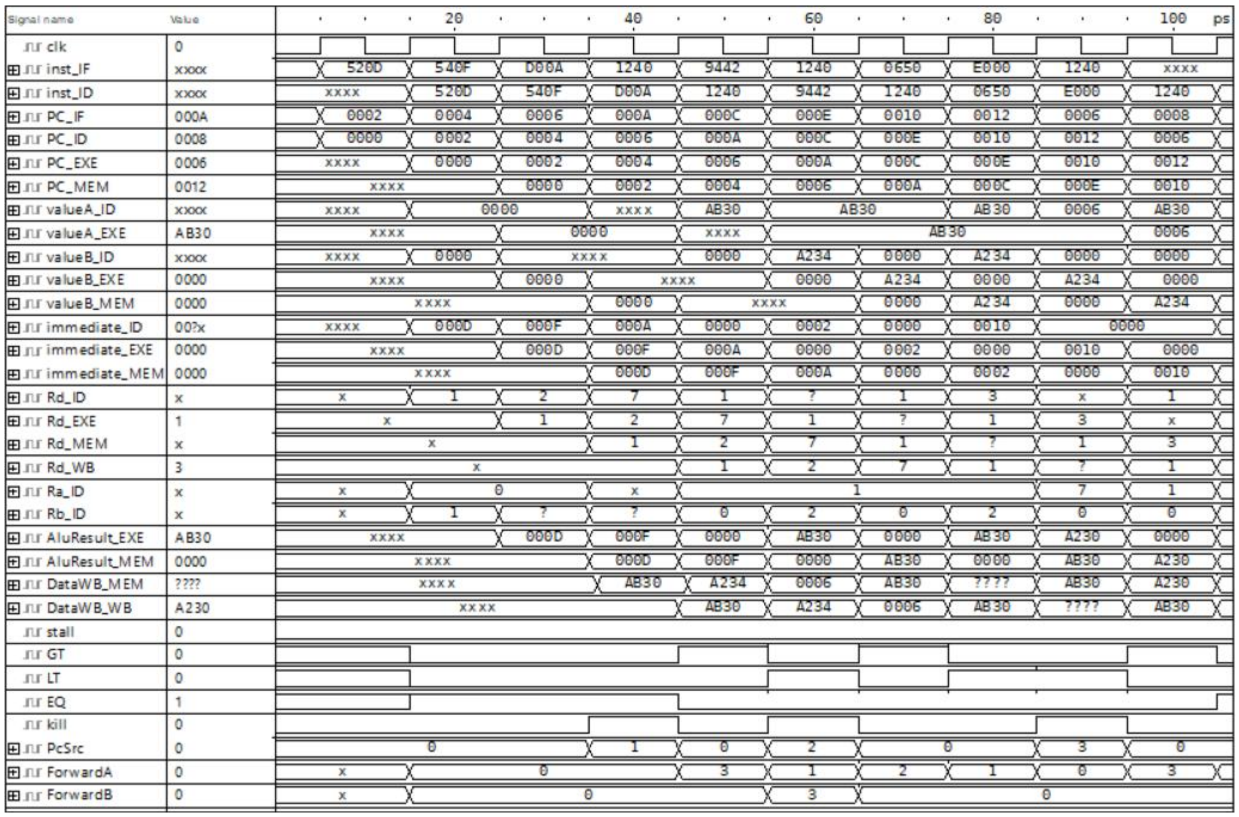


Figure 21 Program 4 Waveform

Explanation

- At second 50, we can notice that R1 was loaded with the value in memory[13:14]
- At second 60, we can notice that R1 was loaded with the value in memory[15:16]
- At second 110, we can notice that the AND operation was indeed done and the result saved in R3.

Teamwork

If it wasn’t for the joint effort of our team, this project wouldn’t have finished on time. The pipelined processor’s complexity requires the teammates each to work on different components concurrently and then for them to test the components they made. After all components were tested, they should combine them for the final top-level module and fix any issues they face. That is exactly what we have done in this project. Mohammad created the fetch stage and it’s components. Ahmad created the execution and memory stages and their components. Qusay created the decode stage. After each teammate was done with their stage, we all joined a discord call and created the top-level processor module. Together, we fixed any issues and debugged any problems we faced.

Conclusion

In this project, we successfully designed and verified a simple pipelined RISC processor using Verilog, demonstrating a comprehensive understanding of computer architecture and digital design principles. Our processor features a five-stage pipeline architecture—fetch, decode, ALU, memory access, and write-back—which efficiently executes a variety of instructions within our defined instruction set architecture (ISA). This ISA supports R-type, I-type, J-type, and S-type instructions, utilizing 8 general-purpose registers and byte-addressable memory.